Simple Obfuscation Assembly Program Report

krandslam

2020/09/13

Table of Contents

1. Make the obfuscation program

```
root@stud: /home/x86_exam/obfus
root@stud:/home/x86_exam/obfus# vi ./obfus.asm
```

Make an assembly file “obfus.asm” and open with vi editor.

```
root@stud: /home/x86_exam/obfus
 1 global main
 2 extern printf
 3
 4 section .data
 5         message db "hello world", 10, 0 ; \n: 10 \0: 0
 6
 7 section .text
 8 main:
 9         push    dword message
10         mov     eax, [esp]
11         mov     ecx, 11 ; 11 characters
12         jmp     obfus
13
14 obfus:
15         xor     [eax+ecx-1], byte 0x52 ; little endian
16         dec     ecx
17         jne     obfus
18         push    dword message
19         call    printf
20
21 end:
22         mov     eax, 1
23         mov     ebx, 0
24         int     80h
```

The basic code explanation follows below, and the code will be reviewed deeply during the debugging process.

Code explanation:

- Line 5: make a string called message in data area and the content is “hello world”.

  Orange rectangle - 10 represents newline (\n) and 0 represent null (\0), which indicates the end of string.

- Line 9: push the string into stack
- Line10: move what esp is pointing to eax
- Line11: move 11 (string length) to ecx (used as counter)
- Line12: jump to obfus label
- Line14: start of obfus label
- Line 15-17: loop of XOR operation for each byte of string
- Line 18: push obfuscated string
- Line 19: print the obfuscated string
- Line 21-24: terminate the program by system call function (exit)

2. Debug the obfuscation program

2-1 Compile and Debug

```
root@stud: /home/x86_exam/obfus
root@stud:/home/x86_exam/obfus# nasm -felf32 obfus.asm && ld -I/lib/ld-linux.so.
2 -lc --entry main obfus.o
root@stud:/home/x86_exam/obfus# ls
a.out  obfus.asm  obfus.o
root@stud:/home/x86_exam/obfus#
```

- Red – assembly (assembly code -> object file)
- Blue – link (object file -> executable file)

  "-I/lib/ld-linux.so.2 -lc" is added due to printf function
- Green – executable file in linux.

Now run the program,

```
root@stud: /home/x86_exam/obfus
root@stud:/home/x86_exam/obfus# ./a.out
:7>>=r%= >6
root@stud:/home/x86_exam/obfus#
```

- Red – output of obfuscated string
- We can see that "hello world" is obfuscated to ":7>>=r%= >6"

To better understand what actually happened, we use gdb to debug the program.

```
root@stud: /home/x86_exam/obfus
root@stud:/home/x86_exam/obfus# gdb -q ./a.out
Reading symbols from ./a.out...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
   0x08048190 <+0>:     push   0x8049270
   0x08048195 <+5>:     mov    eax,DWORD PTR [esp]
   0x08048198 <+8>:     mov    ecx,0xb
   0x0804819d <+13>:    jmp    0x804819f <obfus>
End of assembler dump.
(gdb) disas obfus
Dump of assembler code for function obfus:
   0x0804819f <+0>:     xor    BYTE PTR [eax+ecx*1-0x1],0x52
   0x080481a4 <+5>:     dec    ecx
   0x080481a5 <+6>:     jne    0x804819f <obfus>
   0x080481a7 <+8>:     push   0x8049270
   0x080481ac <+13>:    call   0x8048180 <printf@plt>
End of assembler dump.
(gdb) disas end
Dump of assembler code for function end:
   0x080481b1 <+0>:     mov    eax,0x1
   0x080481b6 <+5>:     mov    ebx,0x0
   0x080481bb <+10>:    int    0x80
End of assembler dump.
(gdb)
```

- Red – run the gdb of the executable file a.out

  -q: "Quiet" Do not print the introductory and copyright messages.
- Orange – set gdb to use intel disassembly style
- Blue- disassemble each function

2-2 Debug main function

Let's first examine main function.

```
(gdb) disas main
Dump of assembler code for function main:
   0x08048190 <+0>:     push   0x8049270
   0x08048195 <+5>:     mov    eax,DWORD PTR [esp]
   0x08048198 <+8>:     mov    ecx,0xb
   0x0804819d <+13>:    jmp    0x804819f <obfus>
End of assembler dump.
(gdb) x/x 0x8049270
0x8049270:      0x6c6c6568
(gdb) x/s 0x8049270
0x8049270:      "hello world\n"
(gdb)
```

- The first operation is push 0x8049270 to stack, but we do not know what that value is.
- So, we use x/x to see its hex value.

  0x6c6c6568 is not clear too.
- We use x/s to see its string

  "hello world\n" is there.

  This matches the line 5,9 of assembly code.

Let's set up break point and debug the program.

```
(gdb) b main
Breakpoint 1 at 0x8048190
(gdb) r
Starting program: /home/x86_exam/obfus/a.out

Breakpoint 1, 0x08048190 in main ()
(gdb) disas main
Dump of assembler code for function main:
=> 0x08048190 <+0>:     push   0x8049270
   0x08048195 <+5>:     mov    eax,DWORD PTR [esp]
   0x08048198 <+8>:     mov    ecx,0xb
   0x0804819d <+13>:    jmp    0x804819f <obfus>
End of assembler dump.
(gdb) x/4x $esp         esp           esp+4         esp+8         esp+12
0xbffff0a0:     0x00000001      0xbffff29d      0x00000000      0xbffff2b8
```

- Red – set up break point at start of the main function
- Blue – run the program

- Orange – shows the current state of debugging process
- Green – print 4 word-sized hex values starting from esp.

Now, let's run next instruction.

```
(gdb) ni
0x08048195 in main ()
(gdb) disas main
Dump of assembler code for function main:
   0x08048190 <+0>:     push   0x8049270
=> 0x08048195 <+5>:     mov    eax,DWORD PTR [esp]
   0x08048198 <+8>:     mov    ecx,0xb
   0x0804819d <+13>:    jmp    0x804819f <obfus>
End of assembler dump.
(gdb) x/4x $esp
0xbffff09c:     0x08049270      0x00000001      0xbffff29d      0x00000000
```

- Red – next instruction
- Blue – we can see that current state is shifted
- Orange – The value that is pushed into stack appears in esp.

Before we see the result of next instruction, please take a guess what eax register would have.

I will provide the values of eax and esp register before next instruction.

```
(gdb) info reg $esp
esp            0xbffff09c       0xbffff09c
(gdb) info reg $eax
eax            0x1c     28
```

Now let's check what eax register would have.

```
(gdb) ni
0x08048198 in main ()
(gdb) disas main
Dump of assembler code for function main:
   0x08048190 <+0>:     push   0x8049270
   0x08048195 <+5>:     mov    eax,DWORD PTR [esp]
=> 0x08048198 <+8>:     mov    ecx,0xb
   0x0804819d <+13>:    jmp    0x804819f <obfus>
End of assembler dump.
(gdb) info reg $eax
eax            0x8049270        134517360
(gdb)
```

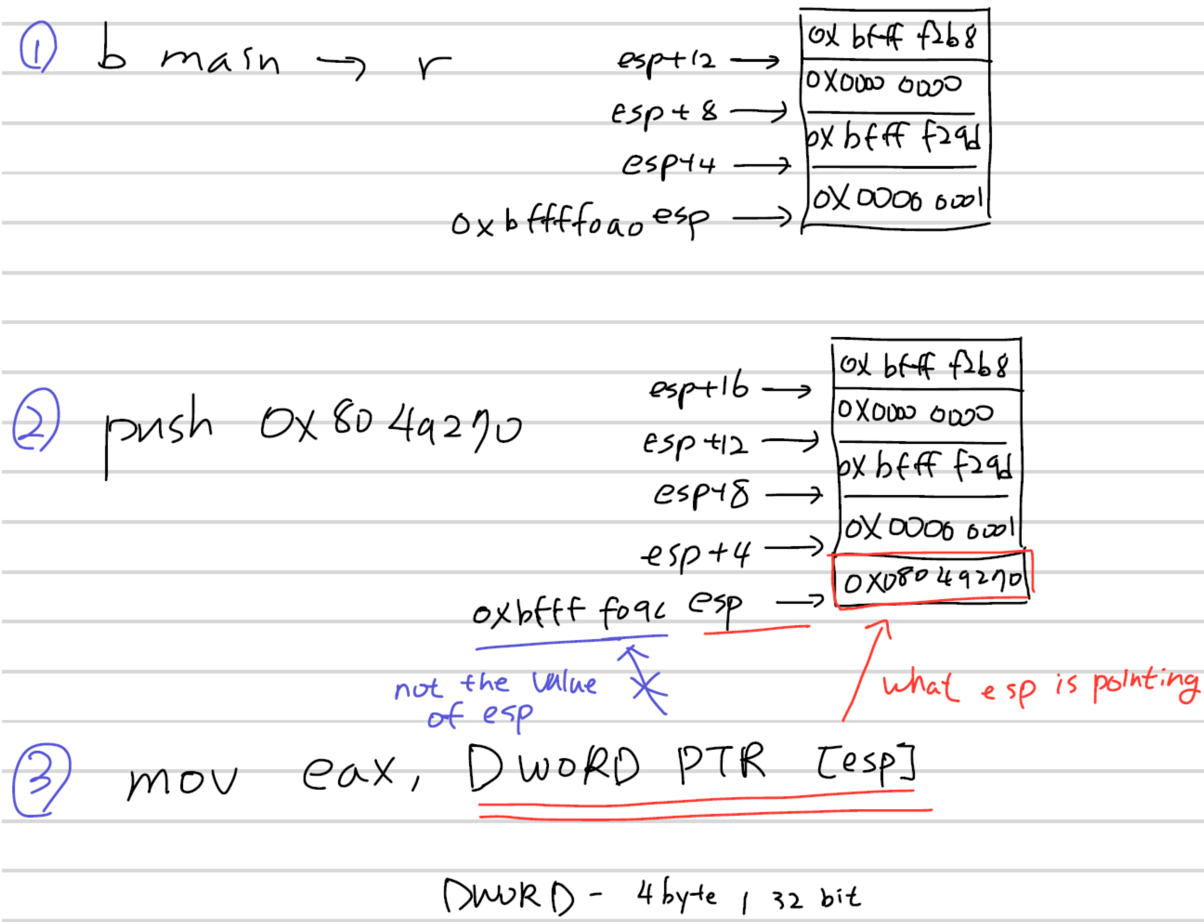
Some could guess that 0xbffff09c, the value of $esp, would move to eax.

But, mov instruction moved 0x8049270("hello world\n") to eax.

DWORD PTR [esp] indicates the value $esp is pointing and load DWORD size (32-bit)

So, the value what $esp is containing moved to eax.

To better understand this, let me draw a stack.



① Shows the stack at the start of main function.

The values in stack were found by x/4x $esp

② The stack grew downward and 0x8049270 is pushed.

The value of esp also decreased.

③ DWORD PTR [esp] is what esp is pointing, not the value of esp.

The corresponding value is moved to eax.

Now next part is straight forward.

```
(gdb) info reg $ecx
ecx            0xb7fffc24       -1207960540
(gdb) ni
0x0804819d in main ()
(gdb) disas main
Dump of assembler code for function main:
   0x08048190 <+0>:     push   0x8049270
   0x08048195 <+5>:     mov    eax,DWORD PTR [esp]
   0x08048198 <+8>:     mov    ecx,0xb
=> 0x0804819d <+13>:    jmp    0x804819f <obfus>
End of assembler dump.
(gdb) info reg $ecx
ecx            0xb      11
(gdb)
```

- Red - 0xb is moved to ecx register

  0xb (11) is length of string (“hello world”)

- Orange – the next instrucion is jump to “obfus” function.

  We can check that below.

```
(gdb) ni
0x0804819f in obfus ()
(gdb) disas obfus
Dump of assembler code for function obfus:
=> 0x0804819f <+0>:     xor    BYTE PTR [eax+ecx*1-0x1],0x52
   0x080481a4 <+5>:     dec    ecx
   0x080481a5 <+6>:     jne    0x804819f <obfus>
   0x080481a7 <+8>:     push   0x8049270
   0x080481ac <+13>:    call   0x8048180 <printf@plt>
End of assembler dump.
(gdb)
```

2-3 Debug obfus function

Before debugging the first instruction, we should understand what BYTE PTR [eax+ecx*1-0x1] means.

We load a byte size from [eax+ecx*1-0x1].

```
(gdb) info reg $eax
eax            0x8049270        134517360
(gdb) info reg $ecx
ecx            0xb      11
(gdb) 
```

[eax+ecx*1-0x1] = [0x8049270+ 0xb*1 -1] = [0x804927a]

Let's see what byte sized value is at 0x804927a.

```
(gdb) x/bx 0x804927a
0x804927a:      0x64
(gdb) x/s 0x804927a
0x804927a:      "d\n"
(gdb) 
```

- Red – the byte-size value of 0x804927a is 0x64.

  According to ASCII code, 0x64 is 'd'.
- Orange – If we check the string, we can check 'd'.

In next instructinos, ecx decrease by 1 from "dec ecx".

jne instruction jumps if ZF = 0. (ZF changes 0 to 1 if ecx = 0)

Therefore, ecx keep decreasing by 1 in this loop until 11 -> 0.

After kowing this, we can conlcude that [eax+ecx*1-0x1] will have a range of [0x804927a] ~ [0x8049270]

```
(gdb) x/s 0x8049279
0x8049279:      "ld\n"
(gdb) x/s 0x8049278
0x8049278:      "rld\n"
(gdb) x/s 0x8049277
0x8049277:      "orld\n"
(gdb) x/s 0x8049276
0x8049276:      "world\n"
(gdb) x/s 0x8049271
0x8049271:      "ello world\n"
(gdb) x/s 0x8049270
0x8049270:      "hello world\n"
(gdb) 
```

- We can check that each byte is the each character of the string.

Now, let's go back to the first instruction.

The first instruction is xor BYTE PTR [0x804927a ~ 0x8049270], 0x52

Then we are doing xor operation with corresponding ASCII value of each character and 0x52.

For example, BYTE PTR [0x804927a] is 'd' and its ASCII value is 0x64.

xor 0x64, 0x52 = 0x36 and saved at left operand "BYTE PTR [0x804927a]".

So, 0x64 at 0x804927a is replaced with value of 0x36.

Let's check this out.

```
(gdb) ni
0x080481a4 in obfus ()
(gdb) disas obfus
Dump of assembler code for function obfus:
   0x0804819f <+0>:     xor    BYTE PTR [eax+ecx*1-0x1],0x52
=> 0x080481a4 <+5>:     dec    ecx
   0x080481a5 <+6>:     jne    0x804819f <obfus>
   0x080481a7 <+8>:     push   0x8049270
   0x080481ac <+13>:    call   0x8048180 <printf@plt>
End of assembler dump.
(gdb) x/bx 0x804927a
0x804927a:      0x36
(gdb) x/s 0x804927a
0x804927a:      "6\n"
(gdb)
```

- The value at 0x804927a changed to 0x36 as expected.
- 0x36 is '6' so, 'd' is obfuscated to '6' after xor operation.

Let’s check next instruction.

```
(gdb) ni
0x080481a5 in obfus ()
(gdb) disas obfus
Dump of assembler code for function obfus:
   0x0804819f <+0>:     xor    BYTE PTR [eax+ecx*1-0x1],0x52
   0x080481a4 <+5>:     dec    ecx
=> 0x080481a5 <+6>:     jne    0x804819f <obfus>
   0x080481a7 <+8>:     push   0x8049270
   0x080481ac <+13>:    call   0x8048180 <printf@plt>
End of assembler dump.
(gdb) info reg $ecx
ecx            0xa      10
(gdb) info reg $eflags
eflags         0x206    [ PF IF ]
(gdb)
```

- ecx decreased to 10 (11 -> 10)
- eflag register shows current state of processor.
- What we are interested is ZF (zero flag). ZF = 1 if ecx = 0.

  ecx = 10, so ZF = 0 so ZF did not appear.

jne 0x804819f means if ZF = 0, jump to 0x804819f, which is start of obfus function.

```
(gdb) ni

Breakpoint 1, 0x0804819f in obfus ()
(gdb) disas obfus
Dump of assembler code for function obfus:
=> 0x0804819f <+0>:     xor    BYTE PTR [eax+ecx*1-0x1],0x52
   0x080481a4 <+5>:     dec    ecx
   0x080481a5 <+6>:     jne    0x804819f <obfus>
   0x080481a7 <+8>:     push   0x8049270
   0x080481ac <+13>:    call   0x8048180 <printf@plt>
End of assembler dump.
```

So, we figured out that this loop keep decreasing ecx by 1 and it will continue until ZF = 1, ecx = 0.

The loop will go through [eax+(11 to 1)-1] , (11 to 1) represents the range of ecx decimal value by the loop.

[eax+(11 to 1)-1] = [0x8049270 +(11 to 1)-1] = [0x8049270 + (10 to 0)]

= [0x804927a to 0x8049270] -> each character of “hello world”

Eventually, we can xor all the characters of string from loop.

The result follows below.

```
(gdb) delete 1
(gdb) b *obfus+8
Breakpoint 2 at 0x80481a7
(gdb) c
Continuing.

Breakpoint 2, 0x080481a7 in obfus ()
(gdb) disas obfus
Dump of assembler code for function obfus:
   0x0804819f <+0>:     xor    BYTE PTR [eax+ecx*1-0x1],0x52
   0x080481a4 <+5>:     dec    ecx
   0x080481a5 <+6>:     jne    0x804819f <obfus>
=> 0x080481a7 <+8>:     push   0x8049270
   0x080481ac <+13>:    call   0x8048180 <printf@plt>
End of assembler dump.
(gdb)
```

- Red – first, delete the previous breakpoint (b main)
- Orange – set new breakpoint at the end of loop

Then, we continue to reach the breakpoint.

Now we are curious about what we are pushing and need to check whether ecx is 0.

```
(gdb) info reg $ecx
ecx            0x0      0
(gdb) x/s 0x8049270
0x8049270:      ":7>>=r%= >6\n"
(gdb)
```

- Red – ecx is indeed 0.
- Orange – this is obfuscated string by xor operation and this string corresponds to the string we saw when we ran exectuable file a.out.

To better understand obfuscated string, I drew the obfuscation process.

Original string	h	e	l	l	o	space	w	o	r	l	d
hex value (ASCII)	0x68	0x65	0x6c	0x6c	0x6f	0x20	0x77	0x6f	0x72	0x6c	0x64
				XOR	0x52						
hex value after XOR	0x3a	0x37	0x3e	0x3e	0x3d	0x72	0x25	0x3d	0x20	0x3e	0x36
obfuscated string	:	7	>	>	=	r	%	=	space	>	6

- XOR operation for each character in original string obtains same result as the obfuscated string that we obtained from debugging.
- The value that XOR with, 0x52 in this case, is randomly selected.

  0x52 can be replaced with any number.

Next, obfuscated string is pushed into stack and printed by printf function.

2-4 Terminate the program (end function)

After calling printf function, the program reaches to end function.

I am just going to explain what each instruction represents.

```
(gdb) disas end
Dump of assembler code for function end:
   0x080481b1 <+0>:     mov    eax,0x1
   0x080481b6 <+5>:     mov    ebx,0x0
   0x080481bb <+10>:    int    0x80
End of assembler dump.
(gdb)
```

- Blue – In linux, this represents system call.
- Red – eax represents system call number and 1 is number for exit function
- Orange – ebx is argument of exit function

So those 3 instruction represents exit(0), and exit(0) terminates the program properly.

3. Obfuscation recovery

3-1 Make the recovery program

So, from the previous section, we learned that XOR operation can obfuscate the original string. But, how can we recover original string from obfuscated string?

We can find the answer from the property of XOR operation.

original string	h	e	l	l	o	space	w	o	r	l	d
hex value (ASCII)	0x68	0x65	0x6c	0x6c	0x6f	0x20	0x77	0x6f	0x72	0x6c	0x64
				XOR	0x52						
hex value after XOR	0x3a	0x37	0x3e	0x3e	0x3d	0x72	0x25	0x3d	0x20	0x3e	0x36
obfuscated string	:	7	>	>	=	r	%	=	space	>	6

So, now we start from obfuscated string. If we do XOR operation with same value that we used to obfuscation (0x52 in this case) again to each character of obfuscated string, we can recover original string.

ex)

- 0x3a XOR 0x52 = 0x68 (‘h’)
- 0x37 XOR 0x52 = 0x65 (‘e’)
- …
- 0x36 XOR 0x52 = 0x64 (‘d’)

Now let’s add obfuscation recovery part in the program.

```
root@stud:/home/x86_exam/obfus# ls
a.out  obfus.asm  obfus.o
root@stud:/home/x86_exam/obfus# cp obfus.asm recov.asm
root@stud:/home/x86_exam/obfus# ls
a.out  obfus.asm  obfus.o  recov.asm
root@stud:/home/x86_exam/obfus# vi ./recov.asm
```

- Copy obfus.asm to new file recov.asm
- recov.asm will print recovered string
- Then, open recov.asm with editor

```
root@stud: /home/x86_exam/obfus
section .text
main:
        push    dword message
        mov     eax, [esp]
        mov     ecx, 11 ; 11 characters
        jmp     obfus

obfus:
        xor     [eax+ecx-1], byte 0x52 ; little endian
        dec     ecx
        jne     obfus
        push    dword message
        mov     ecx, 11 ; counter for recovery
        jmp     recov

recov:
        xor     [eax+ecx-1], byte 0x52 ; recovery xor
        dec     ecx
        jne     recov
        push    dword message
        call    printf

end:
-- INSERT --                                        29,5          66%
```

Only parts in box are changed.

- Red – call printf to print obfuscated string is gone.

  Instead, retain counter for string to use in recovery and jump to recov label.
- Orange – repeat XOR operation to recover original string.

  Same loop system logic works as the first 3 instruction of obfus label.
- Blue – print the recovered string

```
root@stud:/home/x86_exam/obfus# nasm -felf32 recov.asm && ld -I/lib/ld-linux.so.
2 -lc --entry main recov.o
root@stud:/home/x86_exam/obfus# ./a.out
hello world
```

- After compiling and linking, the program successful prints original string “hello world”.

3-2 Debug the recovery program

Since most of assembly code is similar, we need to check mainly two parts.

- Before entering recov label, is string obfuscated?
- After entering recov label, is string recovered to original string?

Let’s first check if the string is obfuscated by XOR operation.

```
root@stud:/home/x86_exam/obfus# gdb -q ./a.out
Reading symbols from ./a.out...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas obfus
Dump of assembler code for function obfus:
   0x0804819f <+0>:     xor    BYTE PTR [eax+ecx*1-0x1],0x52
   0x080481a4 <+5>:     dec    ecx
   0x080481a5 <+6>:     jne    0x804819f <obfus>
   0x080481a7 <+8>:     push   0x8049284
   0x080481ac <+13>:    mov    ecx,0xb
   0x080481b1 <+18>:    jmp    0x80481b3 <recov>
End of assembler dump.
(gdb) b *obfus+8
Breakpoint 1 at 0x80481a7
(gdb) r
Starting program: /home/x86_exam/obfus/a.out

Breakpoint 1, 0x080481a7 in obfus ()
(gdb)
```

- Red – run gdb of executable program
- Orange – We already know that the first three instructions do XOR operation and push the obfuscated string to stack.

  So, we set breakpoint right after all the string characters are obfuscated.

  Then, we run program.

```
(gdb) disas obfus
Dump of assembler code for function obfus:
   0x0804819f <+0>:     xor    BYTE PTR [eax+ecx*1-0x1],0x52
   0x080481a4 <+5>:     dec    ecx
   0x080481a5 <+6>:     jne    0x804819f <obfus>
=> 0x080481a7 <+8>:     push   0x8049284
   0x080481ac <+13>:    mov    ecx,0xb
   0x080481b1 <+18>:    jmp    0x80481b3 <recov>
End of assembler dump.
(gdb) x/s 0x8049284
0x8049284:       ":7>>=r%= >6\n"
```

- We can see that string "hello world" is obfuscated to ":7>>=r%= >6" by XOR operation as expected.

Now let's check if the string is recovered to original string.

```
(gdb) disas recov
Dump of assembler code for function recov:
   0x080481b3 <+0>:     xor    BYTE PTR [eax+ecx*1-0x1],0x52
   0x080481b8 <+5>:     dec    ecx
   0x080481b9 <+6>:     jne    0x80481b3 <recov>
   0x080481bb <+8>:     push   0x8049284
   0x080481c0 <+13>:    call   0x8048180 <printf@plt>
End of assembler dump.
(gdb) b *recov+8
Breakpoint 2 at 0x80481bb
```

- Let's jump to where obfuscated string finishes XOR operation to recover original string.
- Set up break point right after all the string characters completed XOR operation.

```
(gdb) c
Continuing.

Breakpoint 2, 0x080481bb in recov ()
(gdb) disas recov
Dump of assembler code for function recov:
   0x080481b3 <+0>:     xor    BYTE PTR [eax+ecx*1-0x1],0x52
   0x080481b8 <+5>:     dec    ecx
   0x080481b9 <+6>:     jne    0x80481b3 <recov>
=> 0x080481bb <+8>:     push   0x8049284
   0x080481c0 <+13>:    call   0x8048180 <printf@plt>
End of assembler dump.
(gdb) x/s 0x8049284
0x8049284:      "hello world\n"
(gdb)
```

- c – continue to reach the breakpoint
- We can see that obfuscated string successfully recovered back to original string "hello world" after repeating XOR operation.

4. Conclusion

The obfuscation program successfully obfuscated original string "hello world" to obfuscated string ":7>>=r%= >6" by XOR operation.

The recovery program successfully recovered from obfuscated string ":7>>=r%= >6" to original string "hello world" by repeating XOR operation with same value.

Obfuscation with XOR operation is too simple, so application of further complex obfuscation algorithm is necessary to secure important information.